Instructed late binding

Haeri, Seyed H.; Keir, Paul

Published in:
Proceedings of the 23rd Panhellenic Conference on Informatics (PCI 2019)

DOI:
10.1145/3368640.3368644

Published: 05/12/2019

Document Version
Peer reviewed version

Instructed Late Binding

Seyed H. Haeri (Hossein)
Université catholique de Louvain, Belgium
hossein.haeri@ucl.ac.be

Paul Keir
University of the West of Scotland, UK
paul.keir@uws.ac.uk

Abstract

Integration of a decentralised pattern matching is a technique that enables a recent solution to the Expression Problem. The single former implementation of this technique was in Scala. In this paper, we highlight the C++ implementation of the same technique to solve the Expression Problem in C++. Unlike the former implementation which relies on stackability of Scala `trait`s, this new implementation relies on compile-time metaprogramming for automatic iterative pointer introspection at runtime. That iteration enables late binding using overload resolution, which the compiler is already capable of. The C++ implementation outperforms the Scala one by providing strong static type safety and offering considerably easier usage.

1 Introduction

The Expression Problem (EP) [5, 21, 27] is a recurrent problem in Programming Languages (PLs), for which a wide range of solutions have been proposed. Consider those of Torgersen [25], Odersky and Zenger [14], Swierstra [24], Oliveira and Cook [16], Bahr and Hvitved [2], Wang and Oliveira [28], Haeri and Schupp [13], and Haeri and Keir [10], to name a few. EP is recurrent because it is repeatedly faced over embedding DSLs – a task commonly taken in the PL community. Embedding a DSL is often practised in phases, each having its own Algebraic Datatype (ADT) and functions defined on it. For example, take the base and extension to be the type checking and the type erasure phases, respectively. One wants to avoid recompiling, manipulating, and duplicating one's type checker if type erasure adds more ADT cases or defines new functions on them.

Haeri [8] phrases EP as the challenge of implementing an ADT – defined by its cases and the functions on it – that:

- **E1.** is *extensible in both dimensions*: Both new cases and functions can be added.
- **E2.** provides *weak static type safety*: Applying a function f on a statically[1] constructed ADT term t should fail to compile when f does not cover all the cases in t.
- **E3.** upon extension, forces *no manipulation or duplication* to the existing code.
- **E4.** accommodates the extension with *separate compilation*: Compiling the extension imposes no requirement for repeating compilation or type checking of existing ADTs and functions on them. Compilation and type checking of the extension should not be deferred to the link or run time.

A recent solution to EP (and a generalisation of it called the Expression Families Problem [15]) was given by Haeri and Schupp [13]. Their solution was based on a technique called integration of a decentralised pattern matching (IDPaM). Haeri and Schupp presented their solution in Scala. We deploy the same technique here to solve EP in C++.

Scala and C++ are different in many ways, causing significant differences between the Scala IDPaM and the C++ one. The Scala IDPaM uses type constraints and stackability of `trait`s. The C++ IDPaM uses `template` and macro metaprogramming. An overview of the C++ IDPaM at § 3.

Behind the scenes, the C++ IDPaM employs a one-liner macro for instructing the compiler to perform a new kind of late binding. Dynamic dispatch or late binding is a standard compiler technique used widely in object-oriented languages. Multiple dispatch is when more than one function parameter is late-bound. Both single and multiple dispatch are automatic and happen at runtime. Our late-binding, however, is special because the one-liner expands at compile-time to compiler instructions for ADT traversal and introspection. This is to obtain type information at runtime, which the compiler uses to perform overload resolution.

Our contributions are:

- We implement the C++ IDPaM that offers easier roles[2] to take for solving EP than the Scala IDPaM (§ 7) and that better addresses the EP concerns (§ 4). Our key enabler for the relative ease of use is our one-liner macro for instructed late-binding (§ 5).
- We show that IDPaM can offer **strong** static type safety even in the absence of defaults and that stackability is not a cornerstone of IDPaM.
- We implement Generalised Algebraic Datatypes in C++ using the C++ IDPaM.
- We provide the first manifestation of the roles to be taken for solving EP (§ 6). That insight enables us to systematically review related work § 9.
- We embed a total of 25 DSLs using C++ IDPaM (§ 8).

[1] If the guarantee was for dynamically constructed terms too, we would have called it strong static type safety.

This work is partially funded by the LightKone European H2020 Project under Grant Agreement No. 732505 and partially by the Belgian National Fund for Scientific Research (F.R.S.-FNRS).

PCI 2019, 28–30 Nov. 2019, Nicosia, Cyprus
2019.

[2] Cf. § 6 for the role definitions.

We achieve those because C++ metaprogramming allows us to encode the ADT in a way that: firstly, the ADT and its cases can be **programmatically** queried for one another; and, secondly, one can **programmatically** traverse the cases of an ADT for introspection.

The first point above makes definition of both ADTs and functions on them particularly less involved than the Scala IDPaM. That reduced involvement is significant. Recall that EP is the essence of challenges in embedding DSLs. During the latter exercise, that reduced involvement is a precious gift – especially, with automation of the embedding in mind.

We find the mere implementation of IDPaM in C++ constitutes valuable research. That is because, with the categorical differences between Scala and C++, this work demonstrates independence of IDPaM from Scala. Here are three differences that come to play in this work:

First, a cornerstone in Scala is its dependently typed methods – a feature not available in C++. Second, Scala offers no guarantee for any method call not to be late-bound. On the contrary, in C++, only calls to member functions through references or pointers are late-bound. Third, in the Scala IDPaM, the stackability of `trait`s takes a major role. The closest language feature that C++ offers to `trait`s is multiple inheritance, albeit with no stackability.

Given its presentation in C++, the C++ IDPaM machinery may look as an EP solution that is too specific to C++. We recall, however, that it is typical for EP solutions to be presented with tactful uses of a single language. Take Datatypes à la Carte [24], CDTs [1], PCDTs [2], and MRM [17] in Haskell, Polymorphic Variants [6] in OCaml, and LMS [22] and MVCs [15] in Scala. The C++ IDPaM is amongst the few EP solutions that made it to a mainstream PL.

This paper leaves many details out due to space restrictions. The complete version is available online[3].

2 Existing IDPaM Technology

It is common for EP solutions to take the following as the running example: An ADT *NA* with *N*umbers and *A*ddition as the only cases and another ADT *NAM* with *M*ultiplication in addition to those of *NA*:

$$\begin{array}{lcll} \alpha_{NA} & = & \underline{Num}_{NA}(n) & | \quad \underline{Add}_{NA}(\alpha_{NA}, \alpha_{NA}), \\ \alpha_{NAM} & = & \underline{Num}_{NAM}(n) & | \quad \underline{Add}_{NAM}(\alpha_{NAM}, \alpha_{NAM}) \\ & | & \underline{Mul}_{NAM}(\alpha_{NAM}, \alpha_{NAM}). & \end{array}$$

Using $\gamma\Phi C_0$ [12] as the formalism, one denotes those as: $NA = \underline{Num} \oplus \underline{Add}$ and $NAM = \underline{Num} \oplus \underline{Add} \oplus \underline{Mul}$. Notice how the latter formalism dismisses the subscripts for *Num*, *Add*, and *Mul*. We use the $\gamma\Phi C_0$ notation hereafter.

It is tempting to use plain OOP for solving EP. The idea would be for ADT cases to inherit from the ADT itself and virtual functions to implement the functions defined on the ADT. That way, new cases could be defined by simple new derivations from the ADT. Furthermore, with virtual functions being readily late-bound by the compiler, runtime polymorphism would ensure the correct function behaviour on each ADT case. That approach fails to address E3 and E4 because addition of new functions amounts to recompiling the entire existing hierarchy and the depending code. We now review the Scala IDPaM [13]:

Case Components IDPaM too prescribes for the ADT cases to inherit from the ADT. But, IDPaM is also inspired by Component-Based Software Engineering (CBSE) [23, §17],[19, §10]. Like previous EP solutions of Haeri and Schupp [11, 8, 12], IDPaM takes a *components-for-cases* (C4C) approach: that is, each ADT case is implemented using a standalone component (in its CBSE sense) that is ADT-parameterised, a.k.a., *case components*. Here are the Scala case components for *Num* and *Add*:

```
trait IAE[E <: IAE[E]]
class Num[E <: IAE[E],
          N <: Num[E, N] with E](val n: Int)
class Add[E <: IAE[E],
          A <: Add[E, A] with E](val left: E,
                                 val right: E)
```

ADT Definition Armed with those, one defines *NA* as:

```
trait NA extends IAE[NA]
case class Num(n: Int) extends Num[NA, Num](n) with NA
case class Add(left: NA, right: NA) extends
  Add[NA, Add](left, right) with NA
```

Match Components Instead of virtual functions of OOP, for the implementation of functions defined on a datatype, IDPaM gets the programmer to instruct the late-binding. For example, here is how to obtain pretty-printing for `NA`:

```
object to_string extends PrB[NA] with
  PrN[NA] with PrA[NA] with PrF[NA]
```

`to_string` assembles building blocks provided by the programmer. (Cf. § 5 for the macro details.) In short, the assembled building blocks form a structure akin to the familiar pattern matching of functional programming. The familiar pattern matching, however, is holistic; all the match statements are together in the pattern matching and the individual match statements do not exist elsewhere. One can essentially not detach the individual match statements from the holistic pattern matching. Reusing the match statements is, hence, impossible. What we referred to above as the building blocks are, on the contrary, independent of the resulting assembly in which they set up. Those are, again, components in the CBSE sense. We call them the *match components*. `PrN` and `PrA` above are the pretty-printing match components of *Num* and *Add*:

```
class PrN[E <: IAE[E]] {
  override def the_to_string_match: E => String = {
    case (n: Num[_, _]) => n.toString
    case (e: E)         => super.the_to_string_match(e)}}
class PrA[E <: IAE[E]] {
  override def the_to_string_match: E => String = {
```

[3] See `https://tinyurl.com/yyo8wl4b` and `https://tinyurl.com/yyh8avld`.

```
    case (a: Add[_, _]) => to_string(a.left) + " + " +
        to_string(a.right)
    case (e: E)         => super.the_to_string_match(e)}}
```

And, `PrB` and `PrF` are technical details required at the beginning and at the end of every integration that leads to definition of a function on an ADT.

```
class PrB[E <: IAE[E]] {
  def the_to_string_match: E => String = { throw ... }
  def to_string(e: E): String}
class PrF[E <: IAE[E]] {
  override def to_string(e: E): String =
    super.the_to_string_match(e)}
```

Each match component corresponds to one and only one match statement – enabling *decentralisation* of a pattern matching. The set of match statements and their order, in the Scala IDPaM, is open to the programmer for configuration at the right time. Instead of it being delivered holistically, the pattern matching then is by *integration* of the (decentralised) match components. Hence, the IDPaM name.

3 How to Use the C++ IDPaM

Case Components In the C++ IDPaM, the ADT-parametrisation translates to type-parametrisation by ADT. For example, here are the case components for *Num* and *Add*:

```
1 template<typename ADT> struct Num: ADT //Num α : ℤ → α
2 { Num(int n): n_(n) {} int n_; };
3 template<typename ADT> struct Add: ADT{//Add α : α × α → α
4   Add(const ADT& l, const ADT& r):
5     l_(msfd<ADT>(l)), r_(msfd<ADT>(r)) {}
6   const std::shared_ptr<ADT> l_, r_;
7 };
```

Notice how `Num` and `Add` both take `ADT` as a type parameter and inherit from it. This is a specific way of F-Bounding [4] commonly referred to in C++ as Mixins [26, §21.3].

ADT Definition The necessary steps for implementing *NA* in C++ IDPaM is:

```
1 template<> struct adt_cases<NA>
2 { using type = std::tuple<Num<>, Add<>>; };
```

The above `template` specialisation of `adt_cases` for `NA` (line 1) is a metafunction instructing the compiler for `adt_cases` := `adt_cases` ∪ {`NA` ↦ *Num* ⊕ *Add*}. That is, it introduces `std::tuple<Num<>, Add<>>` to the compiler as the *case list* of `NA`, enabling the compiler to infer the former from the latter, when required. Other case component combinations are done similarly, provided the presence of the additional case components.

Match Components Recall from § 2 that, for the implementation of functions defined on a datatype, IDPaM gets the programmer to instruct the late-binding. In the C++ IDPaM, this instruction is a one-liner that takes advantage of overload resolution – as readily available in C++. That instructed late-binding is a one-off for each function defined on ADTs, no matter how many new case components are added later. For example, the one-off macro expansion for pretty-printing is:

```
1 GENERATE_DISPATCH(string, to_string)
```

Under the hood, the one-liner assembles match components. In the C++ IDPaM, the match components are simply function overloads. For example, the two overloads of `the_to_string_match` below are the two match components for the pretty-printing of `Num` and `Add` above:

```
1 template<typename ADT>//to_string(Num(n)) = to_string(n)
2 string the_to_string_match(const Num<ADT>& n)
3 {return std::to_string(n.n_);}
4 //to_string(Add(l, r)) = to_string(l) + " + " + to_string(r)
5 template<typename ADT>
6 string the_to_string_match(const Add<ADT>& a)
7 {return to_string(*a.l_) + " + " + to_string(*a.r_);}
```

Using some syntactic sugaring, one gets "5 + 5 + 4" for `to_string(5_n + 5_n + 4_n)`, as expected.

4 Addressing The EP Concerns

4.1 E1 (Bidimensional Extensibility)

Adding a new case is a matter of implementing a new case component. For example, a case for *M*ultiplication can be provided by implementing a `Mul` just like `Add` in § 3:

```
1 template<typename ADT> struct Mul: ADT {
2   Mul(const ADT& l, const ADT& r):
3     l_(msfd<ADT>(l)), r_(msfd<ADT>(r)) {}//See § 4.2...
4   const std::shared_ptr<ADT> l_, r_;
5 };
```

A new ADT *NAM* = *Num* ⊕ *Add* ⊕ *Mul* can then be implemented as easy as *NA* in § 3:

```
1 template<> struct adt_cases<NAM>
2 {using type = std::tuple<Num<>, Add<>, Mul<>>;};
```

Note that `NAM` neither replaces nor shadows over `NA`. The above two ADT types are completely independent and can coexist even in the same `namespace`. It only is that, conceptually, `NAM` is a compatible extension [8] to `NA` because all the cases of the latter are also cases of the former. One can take that relationship between `NAM` and `NA` on board as:

```
template<> struct adt_cases<NAM>
{using type =
  tuple_type_cat<typename adt_cases<NA>::type, Mul<>>;};
```

where implementing `tuple_type_cat` for concatenation of types and `std::tuple` types is routine.

Pretty-printing for `Mul` simply adds the pertaining overload for `the_to_string_match` in § 3.

```
1 template<typename ADT>
2 string the_to_string_match(const Mul<ADT>& m)
3 {return to_string(*m.l_) + " * " + to_string(*m.r_);}
```

Note that, upon addition of `Mul`'s match component, there is no need for expanding the one-liner again for `to_string`.

Finally, brand new functions like evaluation on the existing cases take (i) a new macro expansion to instruct late-binding (c.f., § 5) `GENERATE_DISPATCH(int, eval)`; and, (2) implementation of the match components.

```
template<typename ADT>
int the_eval_match(const Num<ADT>& n) {return n.n_;}
template<typename ADT>
int the_eval_match(const Add<ADT>& a)
{return eval(*a.l_) + eval(*a.r_);}
template<typename ADT>
int the_eval_match(const Mul<ADT>& m)
{return eval(*m.l_) * eval(*m.r_);}
```

4.2 E2 (Static Type Safety)

If the programmer forgets to include a case in the case list of an ADT, the compiler will not allow construction of terms using that case for that ADT.[4] Compilation of the expression `12_n * 14_n`, for example, will fail because of the following `static_assert`ion in the body of `Mul`: `static_assert(is_case_in_adt<Mul<>, ADT>::value, ...)`.

Even attempting `using NAMul = Mul<NA>` will fail. That is achieved using SFINAE[5] techniques that we do not present here. If the respective match component of a case is not available (say, because it is forgotten), the code will be rejected at compile-time. For example, if the pretty-printing match component is not provided for `Mul`, the following compile-error will be produced for `NAM`: "no matching function for call to '`the_to_string_match(const Mul<NAM>&)`'."

4.3 E3 (No Manipulation/Duplication)

Notice how nothing in the evidence for our support for E1 and E2 requires manipulation, duplication, or recompilation of the existing codebase. Our support for E3 follows.

4.4 E4 (Separate Compilation)

Our support for E4, in fact, follows just like E3. It turns out, however, that C++ `template`s enjoy two-phase translation [26, §14.3.1]: The parts that depend on the type parameters are type checked (and compiled) only when they are instantiated, i.e., when concrete types are substituted for all their type parameters. As a result, type checking (and compilation) will be redone for every instantiation. That type-checking peculiarity might cause confusion w.r.t. our support for E4.

In order to dispel that confusion, we need to recall that `Add`, for instance, is a class `template` rather than a class. In other words, `Add` is not a type (because it is of kind $* \longrightarrow *$) but `Add<NA>` is. The interesting implication here is that `Add<NA>` and `Add<NAM>` are in no way associated to one another. Consequently, introduction of `NAM` in presence of `NA`, causes no repetition in type checking (or compilation) of `Add<NA>`. (`Add<NAM>`, nonetheless, needs to be compiled in the presence of `Add<NA>`.) The same argument holds for every other case component or match component already instantiated with the existing ADTs.

More generally, consider a base ADT $\Phi_b = \oplus\overline{\gamma}$ and its extension $\Phi_e = (\oplus\overline{\gamma}) \oplus (\oplus\overline{\gamma}')$. Let $\#(\overline{\gamma}) = n$ and $\#(\overline{\gamma}') = n'$, where $\#(.)$ is the number of components in the component combination. So too assume a C++ IDPaM codebase that contains case components for $\gamma_1, \ldots, \gamma_n$ and $\gamma'_1, \ldots, \gamma'_{n'}$. Defining Φ_b in such a codebase incurs compilation of n case components. Defining Φ_e on top incurs compilation of $n + n'$ case components. Nevertheless, that does not disqualify our EP solution because defining the latter component combination does not incur recompilation of the former component combination. Note that individual components differ from their combination. And, E4 requires the combinations not to be recompiled.

Here is an example in terms of DSL embedding. Suppose availability of a type checking phase in a C++ IDPaM codebase. Adding a type erasure phase to that codebase, does not incur recompilation of the type checking phase. Such an addition will, however, incur recompilation of the case components common between the two phases. Albeit, those case components will be recompiled for the type erasure phase. That addition leaves the compilation of the same case components for the type checking phase intact. Hence, our support for E4.

A different understanding from separate compilation is also possible, in which: an EP solution is expected to, upon extension, already be done with the type checking and compilation of the "core part" of the new ADT. Consider extending *NA* to *NAM*, for instance. With that understanding, *Num* and *Add* are considered the "core part" of *NAM*. As such, the argument is that the type checking and compilation of that "core part" should not be repeated upon the extension.

However, before instantiating `Num` and `Add` for `NAM`, both `Num<NAM>` and `Add<NAM>` are neither type checked nor compiled. That understanding, hence, refuses to take our work for an EP solution. We find that understanding wrong because the core of *NAM* is *NA*, i.e., the *Num*⊕*Add* **combination**, as opposed to both *Num* and *Add* individually. Two quotations back our mindset up:

Zenger and Odersky [14] use the term "processors" for what we call "functions on datatypes." Their definition of separate compilation is as follows: "Compiling datatype extensions or adding new processors should not encompass re-type-checking the original datatype or existing processors." Observe how compiling `NAM` does not encompass repetition in the type checking and compilation of `NA`.

Wang and Oliveira [28] say an EP solution should support: "software evolution in both dimensions in a modular way, without modifying the code that has been written previously." Then, they add: "Safety checks or compilation steps must not be deferred until link or runtime." Notice how neither definition of new case components or ADTs, nor addition of case components to existing ADTs to obtain new ADTs, implies modification of the previously written code. Compilation or type checking of the extension is not deferred to link or runtime either. The situation is similar for the definition of new match components.

[4] This applies to both statically **and** dynamically constructed terms; hence, strong static type-safety.

[5] Substitution Failure Is Not An Error [26, §8.4]

For more elaboration on the take of Wang and Oliveira on (bidimensional) modularity, one may ask: If *NA*'s client becomes a client of *NAM*, will the client's code remain intact under E3 and E4? Let us first disregard code that is exclusively written for *NA* and not meant for reuse by *NAM*: `void na_client_f(const NA&){...}`. If on the contrary, the code only counts on the availability of *Num* and *Add*:

```
template <
  typename ADT, typename =
    std::enable_if_t<adt_contains_v<ADT, Num<>, Add<>>>
> void na_plus_client_f(const ADT& x) {...}
```

Then, it can be reused upon transition from *NA* to *NAM*.

5 One-Liner

Before the technical development of this section, we would like to explain a naming intention of ours: We chose the name "one-liner" because it takes only one line of code to **use** the macro `GENERATE_DISPATCH` for instructing the late-binding. The definition of the one-liner, however, takes multiple dozens of lines.

It now is time to reveal the technology behind our one-liner. We begin by presenting some utility macros used by the one-liner macro. The two macros below facilitate name production for the different role players in the one-liner technology. They both do so by juxtaposition of tokens during macro expansion.

Given a token `name`, the macro `FUNC_MATCH(name)` expands to `"the_"` followed by `name` followed by `"_match"`. This is a naming convention in our codebase: For a function `f` on ADTs, the match components need to be called `the_f_match`. For example, note that with the second argument passed to the one-liner macro in § 3 being `to_string`, the match components are called `the_to_string_match`.

The macro `FUNC_DISPATCH(name)` expands similarly to `FUNC_MATCH(name)`. Again, this is a naming convention we follow: For a function `f` on ADTs, an internal set of `template` specialisations will be generated upon expansion of the one-liner macro that are all called `the_f_dispatcher`. Those `template` specialisations are at the core of the automation provided by the one-liner. See § 5.2 for more details.

Recall from § 3 that the one-liner for pretty-printing was called with `to_string` as the second argument. Recall also that the user did not implement `to_string` itself but used it. See line 7 of `the_to_string_match` at the bottom of § 3, e.g., in the match components and `to_string(5_n + 5_n + 4_n)` for concrete expression comparison.

The `to_string` function is the contact point for the user of the dispatcher. All it does is to kick-start the recursive navigation of the relevant ADTs for pointer introspection. Therefore, we call such functions the *call centres*.

5.1 Rewriting Rules of The Dispatcher

It takes seven rewriting rules to formally specify our C++ dispatcher. Here, we only present one rule out of the seven. In § 5.2, we relate the presented (syntactically noisy) metaprogramming snippet for this rule. In our term rewriting, we use the following list comprehension syntax: A list of elements $\overline{e}$ can either be empty (ϵ) or of the form $e_1 \ldots e_n$.

The Dispatcher function takes three type parameters: $\overline{\gamma}_a$, $\overline{\alpha}_x$, and $\overline{\gamma}_x$. It also takes two regular parameters: $\overline{a}$ and $\overline{x}$. Given that, in the metaprogramming, it is the call centre that first invokes the dispatcher, we rather refer to $\overline{a}$ and $\overline{x}$ as the regular *arguments* (passed to the call centre). As such, $\overline{a}$ are the arguments already introspected and $\overline{x}$ are those still to be introspected. Here is the explanation of the three type parameters of the Dispatcher function: $\overline{\gamma}_a$ are the cases that $\overline{a}$ are instances of. $\overline{\alpha}_x$ are the ADTs of $\overline{x}$. And, $\overline{\gamma}_x$ are the remaining cases of $head(\overline{\alpha}_x)$, if any.

Initially, all the (static) type information that is available about $\overline{x}$ is their ADTs. One after the other, arguments in $\overline{x}$ are tested, then, against each and every case in the case lists of their **own** ADTs – unless the test succeeds. The purpose of that test is to figure out whether the argument has also got the (dynamic) case type. (That is, whether the argument is an instance of the case.) Roughly speaking, the test goes on until it either succeeds or the case list of the argument's ADT runs out. In the former situation, the test moves to the next argument in $\overline{x}$, whilst an error is emitted in the latter. What we referred to in the previous paragraph as *test*ing the $\overline{x}$ is a meaning for what we earlier called argument introspection.

When the introspection of an argument x is done, its case type is bookmarked for future in $\overline{\gamma}_a$. Additionally, x itself dynamically cast into its case type is stored in $\overline{a}$. Accordingly, once all the arguments are introspected, all the case types are known and the match component of the right type ($\overline{\gamma}_a$) can be called on correctly cast arguments ($\overline{a}$). The rule *S*#2 manifests that: (The match component is denoted by $m{<}\overline{\gamma}_a{>}$.)

$$\frac{}{\text{Dispatcher}{<}\overline{\gamma}_a, \epsilon, \epsilon{>}(\overline{a}, \epsilon) \longrightarrow m{<}\overline{\gamma}_a{>}(\overline{a})} \ (S\#2)$$

5.2 The C++ Dispatcher

A difference between the rewriting rules and the C++ dispatcher is that type parameters of the latter are type **tuples**; due to a C++ technicality that is beyond us here. We keep the names of the C++ type parameters as close to their term rewriting counterparts as possible. For example, in lines 3–5 below, `GammaAsTup`, `AlphaXsTup`, and `GammaXsTup` are the `tuple`s for $\overline{\gamma}_a$, $\overline{\alpha}_x$, and $\overline{\gamma}_x$, respectively. Lines 2–6 below produce the general `template` signature for the C++ dispatcher. `template` specialisations, then, implement our rewriting rules, one of which is that of *S*#2 (lines 7–12 below).

```
#define GENERATE_DISPATCH(return_type,function_name)  \
template<                                             \
  typename GammaAsTup,                                \
  typename AlphaXsTup,                                \
  typename GammaXsTup                                 \
> struct FUNC_DISPATCH(function_name);                \
```

The patterns we match type arguments against for the rewriting rules imply using partial `template` specialisation. But, C++ does not offer that for functions at the moment – leaving us to `class`es or `struct`s. We choose the latter for there is no need for encapsulation for the four scenarios. Those scenarios, however, need to also have call signatures. For each `template` specialisation, therefore, we provide a `static` member function that is called `match`, by convention. Those `match` member functions (e.g., line 11 below) get called recursively according to the rewriting rules.

```
7  template<typename... GammaAs>                                    \
8  struct FUNC_DISPATCH(function_name)          /* S#2  */\
9    <tuple<GammaAs...>, tuple<>, tuple<>> {  /* γ̄, ϵ, ϵ */\
10     static return_type match(GammaAs... as)                      \
11     { return FUNC_MATCH(function_name)(as...); }                 \
12 };                                                                \
```

The implementation of $S\#2$ starts by pattern matching $\overline{\gamma}_a$, $\overline{\alpha}_x$, and $\overline{\gamma}_x$. According to ($S\#2$), that is matching against $\overline{\gamma}$, ϵ, and, ϵ (line 9). In line 10, it takes $\overline{a}$ as the argument, along with nothing (ϵ). Finally, in line 12, it delegates the call to $m\langle\overline{\gamma}_a\rangle$ (embodied by `FUNC_MATCH(function_name)`), with $\overline{a}$ as the arguments. Note that, due to line 10, the compiler already knows the types $\overline{\gamma}$, with which it picks the right match component out of the available overloads.

6 Roles

When it comes to comparison between different EP solutions, it is natural to choose the proficiency in addressing E1–E4 as the basis. Proficiency, here, is often understood as the relative effort required for addressing those EP concerns. We argue that it is meaningful to also compare EP solutions based on the facilities they offer to each role.

To that end, one needs to first study the roles that are to be taken for an EP solution to develop. The following roles come to mind:

- EP Solver: Brings a particular discipline to the implementation of ADTs and functions defined on them that solves EP.
- Solution User: Implements the ADTs and functions defined on them by submitting to the discipline brought by the EP solver.
- ADT User: Employs the solution user's ADTs and functions on them to create ADT terms and apply functions on those terms.

EP solutions often do not distinguish between those roles.

Whilst exercising the IDPaM discipline, the solution user takes the case components and the match components off-the-shelf. As such, one can imagine two more roles in IDPaM: the case component vendor and the match component vendor. The IDPaM solution user also requires a medium for integrating the match components. In the C++ IDPaM, that medium is the one-liner detailed in § 5, which is provided as a one-off library facility.

7 Comparison with the Scala IDPaM

EP Solver In the Scala IDPaM, ADT cases need to both inherit from the respective case component and the ADT. For example, the *Num* of *NA* in § 2 inherits both from `Num` (the case component) and `NA`. That is the `case class` `Num` in line 2 above. Additionally, due to technicalities of F-Bounding in Scala, the `case class` `Num` requires to tie the recursive type knot by substituting itself for the (second) type parameter of the `Num` case component.

In the Scala IDPaM, the programmer instructs the integration by **manual** assembly of match components in a feature-oriented fashion (cf. `to_string`, § 2).

With components having no definite representative in current programming languages, one needs to leverage other residents of the language to simulate components. In comparison to the Scala IDPaM, in the C++ IDPaM, the discipline required for component development is slightly more lightweight because it requires no type constraints. With C++ being a more verbose language, the Scala implementation is, however, more succinct.

The Scala IDPaM utilises `trait`s for match components. It uses `trait` stackability and `super` calls to enable integration. When a match component of theirs cannot handle the given task, it performs a `super` call, hoping that an upper match component in the stack will pick the task up.

The Scala IDPaM addresses all the EP concerns except E2. It is only in the presence of defaults [29] that it can also address that concern. That is, upon integration, if the programmer forgets to mix-in the respective match component of a case, only a runtime exception will be thrown. This is not totally unacceptable in object-oriented EP solutions, e.g., Lightweight Modular Staging (LMS) [22], MVCs [15], and Torgersen's second solution [25].

On the contrary, in the C++ IDPaM, E2 and the other EP concerns are all addressed. This is an important milestone because it proves that lack of support for E2 is not central to IDPaM. More to the point is IDPaM's capability of providing strong static type safety, even in the absence of defaults.

In comparison to the Scala IDPaM, understanding how the C++ IDPaM comes to be an EP solution is admittedly more difficult. This is because the Scala IDPaM only uses `trait`s: a lay feature of Scala. Our usage of `template` and macro metaprogramming, on the contrary, is relatively advanced in C++. That makes the C++ IDPaM less accessible in comparison to the Scala one.

Solution User Implementing ADTs in the Scala IDPaM is considerably more involved than the C++ IDPaM. The order of invoking match components, in the Scala IDPaM, gets fixed upon the integration. In the C++ IDPaM, on the contrary, the order is unknown to the programmer and the compiler chooses to invoke match components according to the component combination in the term.

A distinctive difference between the C++ and the Scala IDPaM is as follows: In the latter work, for **every** ADT, the programmer is required to assemble the match components – even when the new ADT has the same case list as an existing ADT. For example, for an ADT $NA_2 = Num \oplus Add$, trying the object `to_string` above to terms of NA_2 will fail to compile.

On the contrary, the one-liner macro of the C++ IDPaM is only required to be expanded once for a given function to work for all ADTs ever. Besides, using the case list of a given ADT, the C++ compiler **automatically** assembles the right match components. This facility is available because, in C++, the programmer can define metafunctions too, i.e., functions on types that operate at compile time exclusively.

ADT User Both Scala and C++ facilitate syntax sugaring to a great deal so that using ADTs takes comparable efforts across the two languages.

Component Vendors Match components are simple overloads of `template` functions in the C++ IDPaM. In the Scala IDPaM, on the other hand, match components are type parameterised `trait`s with a method named according to a naming convention that uses Scala's built-in pattern matching and `super` calls. Relying on that built-in support buys extra simplicity for the Scala IDPaM when the match components require other type parametrisation than the ADT they are integrating over. In the C++ IDPaM, we need special treatment for dealing with that. Providing case components takes much less effort despite the syntactic noise of C++.

8 Case Studies

Higher Arity In this case study, we tested our technology against multiple dispatch on functions of higher arity. For example, we had a function *rank* of arity $3 \times 4 \times 2$.

Core Lazy Calculi In this case study, we embedded in C++ a family of eleven big-step operational semantices for lazy evaluation. Multiple dispatch is a key ingredient for implementing formal semantics. Our technology easily accomplished that.

Language Composition In this case study, we focused on composition of DSLs embedded using C++ IDPaM. To that end, we implemented all the sorts of composition presented by Haeri and Keir [9], giving a total of thirteen DSLs.

9 Related Work

Object Algebras Using object algebras [7] to solve EP has become popular over recent years [16, 18, 20, 30, 31]. An often neglected factor about solutions to EP is the complexity of term creation. The symptom develops to the extent that it takes Rendel, Brachthäuser and Ostermann 12 non-trivial lines of code to create a term representing "3 + 5". The ADT user has a considerably more involved job using the current object algebras technologies for EP. The solution user is a slightly heavier role with current object algebras for EP. For example, pretty-printing takes 12 (concise) Scala lines in [20], whereas that is 6 (syntactically noisy) C++ lines in ours. The EP solver's effort, however, is comparable to our solution.

Type Classes Swierstra's Datatypes à la Carte [24] uses Haskell's type classes to solve EP. In his solution too, ADT cases are ADT-independent but ADT-parameterised. He uses Haskell Functors to that end. Defining functions on ADTs amounts to defining a type class, instances of which materialise match statements for their corresponding ADT cases. Without syntactic sugaring, term creation become much more involved than that for ordinary ADTs. As for the EP Solver, because type classes are widely-understood in the programming languages community, Swierstra's discipline is preferable with that background. Ours might be preferable for a mainstream language programmer. For the Solution User, the effort required is comparable across Swierstra's work and that of ours. The ADT User may choose the C++ IDPaM over Swierstra for the syntactic sugaring overhead.

Bahr and Hvitved extend Swierstra's work by offering Compositional Datatypes (CDTs) [1]. Besides, syntactic sugaring is much easier using CDTs because smart constructors can be automatically deduced for terms. CDTs rely on Haskell features that are less widely-understood. EP Solver using CDTs is, therefore, a more complicated role in comparison to that of Swierstra or ours. Solution User is a similar role using CDTs to that using Swierstra's technology. ADT User is comparable of a role with the C++ IDPaM. Later on, Bahr and Hvitved offer Parametric CDTs (PCDTs) [2] for automatic α-equivalence and capture-avoiding variable bindings. Case definitions take two phases: First an equivalent of our case components need to be defined. Then, their case components need to be materialised for each ADT, similar to but different from that of Haeri and Schupp [11, 8]. With their clever Haskell usage and wide list advanced concepts, PCDTs have limited accessibility for the EP Solver. In order to take the benefits of PCDTs over CDTs into action, the Solution User has to put extra effort in, when compared with CDTs. Taking ADT User's role is similar to CDTs.

Modular Reifiable Matching (MRM) [17] improves Swierstra's work by targeting the type class' lack of desirable subtyping between core ADTs and functions defined on them and those of extensions. MRM's use of generic smart constructors makes definition of suitable syntax sugaring trivial.

The distinctive difference between C4C and the works of this group of related work is the former's inspiration by CBSE. Components, in their CBSE sense, ship with their 'requires' and 'provides' interfaces. Whereas, even though the latter works too parametrise cases by ADTs, the interface that CDTs define, for instance, do not go beyond algebraic signatures. Although we do not present that here, C4C goes beyond that, giving easy solutions to the Expression Families Problem [16] and Expression Compatibility Problem [13].

Other Garrigue [6] solves EP using global case definitions that, at their point of definition, become available to every ADT defined afterwards. Per se, a function that pattern matches on a group of these global cases can serve any ADT containing the selected group. OCaml's built-in support for Polymorphic Variants makes all the EP roles easier to take. However, we minimise the drawbacks [3] of ADT cases being global by promoting them to components. That drawback aside, taking all the EP roles is easier in Garrigue's solution.

Rompf and Odersky [22] employ a fruitful combination of the Scala features to present a very simple yet effective solution to EP using LMS. The support of LMS for E2 can be broken using an incomplete pattern matching. Yet, given that pattern matching is dynamic, whether LMS really relaxes E2 is debatable. Ease of taking the EP roles is comparable between LMS and us.

Although not quite related in the techniques used for solving EP, the recent work of Wang and Oliveira [28] is worth noting as well. This latter work is outstanding in its ease of use and the little number of advanced features it expects from the host language.

10 Conclusion

We solve EP in C++ using macro and template metaprogramming for implementing IDPaM. Our solution outperforms its Scala predecessor in that it also statically rejects non-exhaustive function application on an ADT term (constructed statically **or** dynamically), even in the absence of defaults. Furthermore, it is way easier to employ. The cornerstone of the above outperformance is the instructed late-binding. C++ metaprogramming makes that possible by allowing: (1) programmatic queries over an ADT and its case list for one another; and, (2) programmatic traversal of the latter for introspection. Our solution is tested for a set of 25 DSLs.

It turns out that the same technique gives very clean solutions to various generalisations of EP as well as multiple dispatch. Moreover, we believe the use of components in our fashion can make the job of the ADT user considerably easier for object algebras. That is a subject for future research. The material in this paper has thus far only been used for toy examples. It is our aim to benchmark it for embedding medium- and large-scale DSLs.

References

[1] P. Bahr and T. Hvitved. 2011. Compositional Data Types. In 7^{th} *WGP*. Tokyo, Japan, 83–94.

[2] P. Bahr and T. Hvitved. 2012. Parametric Compositional Data Types. In 4^{th} *MSFP (ENTCS)*, J. Chapman and P. B. Levy (Eds.), Vol. 76. 3–24.

[3] A. P. Black. 2015. The Expression Problem, Gracefully. In *MASPEGHI@ECOOP 2015*, M. Sakkinen (Ed.). ACM, 1–7.

[4] P. Canning, W. R. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. 1989. F-Bounded Polymorphism for Object-Oriented Programming. In 4^{th} *FPCA*. 273–280.

[5] W. R. Cook. 1990. Object-Oriented Programming Versus Abstract Data Types. In *FOOL (LNCS)*, J. W. de Bakker, W. P. de Roever, and G. Rozenberg (Eds.), Vol. 489. Noordwijkerhout (Holland), 151–178.

[6] J. Garrigue. 2000. Code Reuse through Polymorphic Variants. In *FSE*. 93–100.

[7] J. V. Guttag and J. J. Horning. 1978. The Algebraic Specification of Abstract Data Types. *Acta Informatica* 10 (1978), 27–52.

[8] S. H. Haeri. 2014. *Component-Based Mechanisation of Programming Languages in Embedded Settings*. Ph.D. Dissertation. STS, TUHH, Germany.

[9] S. H. Haeri and P. Keir. 2019. Composition of Languages Embedded in Scala. In 14^{th} *FedCSIS*, M. Ganzha, L. Maciaszek, and M. Paprzycki (Eds.). IEEE, 209–220.

[10] S. H. Haeri and P. W. Keir. 2019. Metaprogramming as a Solution to the Expression Problem. (Nov. 2019). available online.

[11] S. H. Haeri and S. Schupp. 2013. Reusable Components for Lightweight Mechanisation of Programming Languages. In 12^{th} *SC (LNCS)*, W. Binder, E. Bodden, and W. Löwe (Eds.), Vol. 8088. Springer, 1–16.

[12] S. H. Haeri and S. Schupp. 2016. Expression Compatibility Problem. In 7^{th} *SCSS (EPiC Comp.)*, J. H. Davenport and F. Ghourabi (Eds.), Vol. 39. EasyChair, 55–67.

[13] S. H. Haeri and S. Schupp. 2017. Integration of a Decentralised Pattern Matching: Venue for a New Paradigm Intermarriage. In 8^{th} *SCSS (EPiC Comp.)*, M. Mosbah and M. Rusinowitch (Eds.), Vol. 45. EasyChair, 16–28.

[14] M. Odersky and M. Zenger. 2005. Independently Extensible Solutions to the Expression Problem. In *FOOL*.

[15] B. C. d. S. Oliveira. 2009. Modular Visitor Components. In 23^{rd} *ECOOP (LNCS)*, Vol. 5653. Springer, 269–293.

[16] B. C. d. S. Oliveira and W. R. Cook. 2012. Extensibility for the Masses – Practical Extensibility with Object Algebras. In 26^{th} *ECOOP (LNCS)*, Vol. 7313. Springer, 2–27.

[17] B. C. d. S. Oliveira, S.-C. Mu, and S.-H. You. 2015. Modular Reifiable Matching: A List-of-Functors Approach to Two-Level Types. In 8^{th} *Haskell*. 82–93.

[18] B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook. 2013. Feature-Oriented Programming with Object Algebras. In 27^{th} *ECOOP (LNCS)*, Giuseppe Castagna (Ed.), Vol. 7920. Springer, Montpellier, France, 27–51.

[19] R. S. Pressman. 2009. *Software Engineering: A Practitioner's Approach* (7^{th} ed.). McGraw-Hill.

[20] T. Rendel, J. I. Brachthäuser, and K. Ostermann. 2014. From Object Algebras to Attribute Grammars. In 28^{th} *OOPSLA*, A. P. Black and T. D. Millstein (Eds.). ACM, 377–395.

[21] J. C. Reynolds. 1975. User-Defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction. In *New Direc. Algo. Lang.*, S. A. Schuman (Ed.). INRIA, 157–168.

[22] T. Rompf and M. Odersky. 2010. Lightweight Modular Staging: a Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In 9^{th} *GPCE*. ACM, Eindhoven, Holland, 127–136.

[23] I. Sommerville. 2011. *Software Engineering* (9^{th} ed.). Addison-Wesley.

[24] W. Swierstra. 2008. Data Types à la Carte. *JFP* 18, 4 (2008), 423–436.

[25] M. Torgersen. 2004. The Expression Problem Revisited. In 18^{th} *ECOOP (LNCS)*, M. Odersky (Ed.), Vol. 3086. Oslo (Norway), 123–143.

[26] D. Vandevoorde, N. M. Josuttis, and D. Gregor. 2017. *C++ Templates: The Complete Guide* (2^{nd} ed.). Addison Wesley.

[27] P. Wadler. 1998. The Expression Problem. (Nov. 1998). Java Genericity Mailing List.

[28] Y. Wang and B. C. d. S. Oliveira. 2016. The Expression Problem, Trivially!. In 15^{th} *Modularity*. ACM, New York, NY, USA, 37–41.

[29] M. Zenger and M. Odersky. 2001. Extensible Algebraic Datatypes with Defaults. In 6^{th} *ICFP*. ACM, Florence, Italy, 241–252.

[30] H. Zhang, Z. Chu, B. C. d. S. Oliveira, and T. van der Storm. 2015. Scrap Your Boilerplate with Object Algebras. In 29^{th} *OOPSLA*. 127–146.

[31] H. Zhang, H. Li, and B. C. d. S. Oliveira. 2017. Type-Safe Modular Parsing. In 10^{th} *SLE*. 2–13.